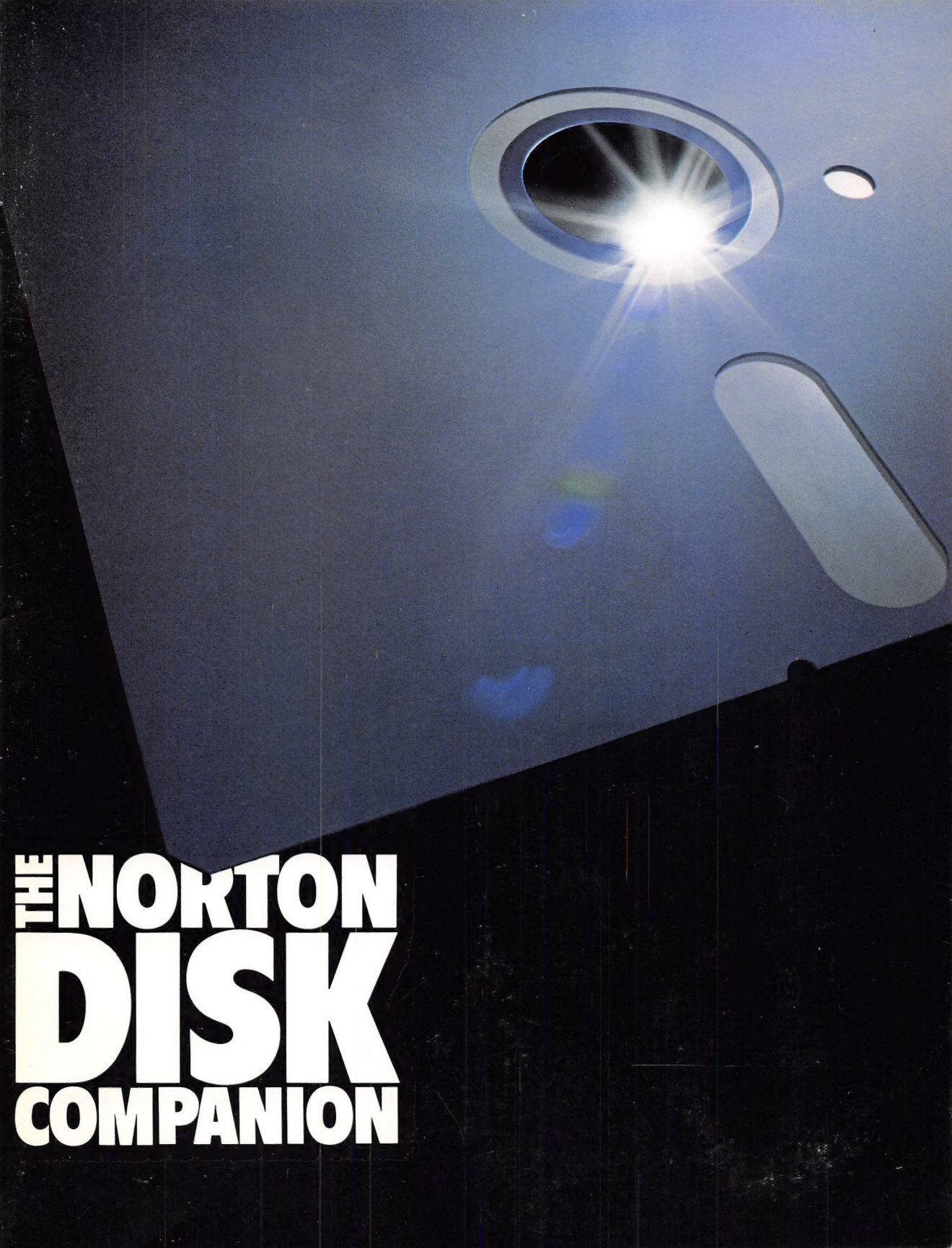

THE NORTON
DISK
COMPANION

The Norton Disk Companion

A Guide to Understanding Your Disks

Introduction

This booklet is a brief tour through the inner workings of disks. We'll begin by examining disks and disk drives from a hardware point of view: how they're made, how they work, what the various parts are called. We'll look at the basic "building blocks," such as platters, heads, tracks, cylinders—what they are, how they work, and how they're put together to make a disk drive.

In the second part of the booklet, we'll concentrate on the software that makes disks work. We'll learn about the difference between physical and logical formatting; look closely at clusters, the basic building blocks of files; examine the infamous File Allocation Table (FAT); check out a number of other software factors that affect disk performance; and conclude with a discussion of disk directories and some of the concepts behind UnErasing.

This second section of the booklet, while occasionally referring back to ideas discussed in the first section, is more or less independent. If you're impatient, therefore, you may want to scan the first section and then go directly to the section on software.

Disks are, to me, one of the most interesting parts of a personal computer. It's a big subject, so I've tried to pick out just those parts of it that you'll find most useful in your day-to-day work. There are several other related topics—disk management, for example—that are interesting and worthy of examination also, but including them would have diluted the sharp focus of this little booklet. To paraphrase the famous tag line from Dragnet: "This booklet is the facts, and just the facts."

I hope you enjoy reading it as much as I enjoyed writing it.

The Hardware

Considering the reputation for complexity that disks—particularly hard disks–seem to have, it's surprising how simple the basic concepts behind them really are.

Although you've probably heard it said a score of times, it bears repeating that disks–both floppy and hard–depend on the same phenomenon as audio or video tape recorders to store their data: A recording head magnetizes microscopic particles embedded in a surface; moving the particles past the magnetized head causes the particles to become magnetized.

In an audio tape—and a digital computer tape too, for that matter—the magnetic medium is simply a long string of plastic tape embedded with metal particles. (Incidentally, the most popular metal particle among tape makers is iron oxide, otherwise known as rust. Floppy disk surfaces are brown because they're covered with rust.) But there's nothing magic about that long thin shape. You could successfully create a magnetic recorder with a medium of any shape, provided you could master the mechanics of moving the medium past a recording head.

Tracks

Imagine touching your index finger to an ink pad and then holding it just above a record spinning on a turntable. If you were to touch the spinning record lightly with your inky finger, you'd leave a finger-width ring of ink on the record. Now imagine that the record is a floppy disk and your fingertip is a magnetic read-write head. The inky trail your finger left on the disk would then have an official name: It would be called a *track*.

By applying a digital signal to the read-write (recording) head, we can record information on the

floppy disk track exactly the same way we could record it on a stringy audio tape. The only difference is that, with an audio tape, we would record a continuously variable signal representing a sound waveform; on a floppy disk we'd record either a maximum signal or nothing at all. In other words, on an audio tape we'd record an *analog* signal, while on a disk we'd record a *digital* signal.

Back on the circular track on our floppy disk, the physics of magnetic heads dictates that the tracks be fairly thin. That means our single track of data on the disk occupies only a fraction of the total disk area. But just as you can record two, four, or more tracks on a single audio tape, so is it possible to record multiple tracks on a single disk. And just as with audio tapes, we can record multiple tracks in either of two ways: by adding more heads, each positioned to record its track inside (with a smaller radius than) the previous one, or by moving a single head back and forth across the disk.

The first method is expensive but fast, since it allows you to record multiple tracks simultaneously. The second method—stepping a single head across the disk—is slower but far more economical, and it's the method most manufacturers have opted for. (But not all: Some high-performance drives do use multiple heads per disk surface as a way to achieve speed.)

Double-Sided Disks

So far, we've considered only one surface of a floppy disk. But just like records, floppy disks have both a top side and a bottom side, and there's nothing to prevent disk manufacturers from treating both to a magnetic coating, allowing us to record on both the top and bottom surfaces. (In that respect, disks are a bit different from tape, which is coated and recordable on only one side. The opposite is actually true of floppy disks: A disk coated with magnetic material on only one side tends to warp.)

Recording on both sides of a disk yields economies of scale. A disk drive that records on a single side needs

a motor to spin the disk, an apparatus to hold the head against the disk, a second motor to move the head back and forth across the disk, a chassis to hold the whole works, and so on. To record on the flip side doesn't require the addition of yet another chassis, motor, or much of anything else. All it requires is a tweak to the apparatus that holds the head—it now becomes a pincer-like affair—and the addition of a second head to the bottom fork of the pincer.

Cylinders

There's another benefit to recording on both sides of a disk: Twice as much data can be written before the head has to be moved from one track to the next. Data can be written first to the track on the top side of the disk; then, without movement of the head, more data can be written to the track on the bottom side. The pair of tracks that lie over each other and can be written without movement of the head assembly are collectively referred to as a cylinder.

For convenient reference, both cylinders and tracks are numbered. The outermost track is called track 0; the track on the top side of the disk is called track 0, side 0, and the track on the bottom is called track 0, side 1. Or you can refer to both track 0s together as cylinder 0. Standard 360K PC floppy disks have 40 cylinders, numbered from cylinder 0 (nearest the rim) to cylinder 39 (nearest the spindle). (When working with disks, we'll find that numbering generally starts at 0, rather than 1. An exception to this is sectors, whose numbering starts at 1. Clusters are another exception, and a curious one at that: Their numbering starts at 2. We'll discuss clusters shortly.)

Let's finish our tour inside a disk drive. So far, we've mostly been describing the double-sided, two-headed floppy disk drive in fine detail. On a floppy disk, the recording head actually sits directly on the disk's magnetic coating. While this allows the head to read and write the strongest signal possible (the strength of the signal falls off quickly as the distance between the head and the magnetic surface in-

creases), it has some drawbacks. Friction, for example, limits the speed at which the disk can spin. Then there's the vexing problem of keeping the head in constant contact with the flexible disk surface, and the equally vexing problem of designing a head that doesn't try to imitate a chisel. Since the speed of rotation directly affects the reading speed—you can't read bits off the disk any faster than they rotate by the head—floppy disks are thus slow by their very nature.

An alternative approach—the one adopted by hard disks—would be to hold the head slightly *above* the disk, so that friction between the head and disk, minor irregularities in the disk surface, and wearing of the disk surface by the head all cease to be problems. A whole other set of problems arises with this approach, however. To start with, since the strength of the signal read or recorded to the disk decreases exponentially as the distance between head and disk increases, it's imperative that we keep the head as close to the disk as possible, yet keep it far enough away so that we're in no danger of touching it—either as components heat up and expand or because of imperfections in the disk surface.

The hard disk designer's rather elegant way of meeting these requirements has been to turn the recording head into a miniature airplane—or, more accurately, a miniature glider.

The hallmark of a hard disk system is the layer of air whipped up by the rapidly spinning disk that the head actually floats on. As you can imagine, that layer of air is very thin. At about 10 microinches (10 millionths of an inch), it doesn't even raise the head enough for a human hair to slide between the bottom of the head and disk surface.

While that's mostly an advantage, since we want to keep the head as close to the surface as possible without touching it, it does preclude the use of a floppy-disk-type flexible backing for the magnetic coating; it simply wouldn't do to have a floppy disk

spinning at 3600 rpm, waving around and flapping into the head a few microinches above it.

The key element of a hard disk, then, is a magnetic head gliding a few microinches above a very rigid disk surface. Because the head must be kept extremely close to the surface, it becomes imperative to post a no-trespassing sign to particles that might otherwise lodge between the head and the disk surface, lest the glider crash into the particle and land on the disk, taking a large byte out of your data. That, in turn, requires sealing the entire head-disk assembly (HDA) and microfiltering the air within the chamber. The requirement to keep fine junk out of the HDA necessitates that that part of a hard disk always be assembled in a clean room, a factor that permanently raises the price floor of hard disks.

With its clean-room assembly, so much of the cost of a hard disk comes in manufacturing the HDA that it makes sense to squeeze as much in there as possible. And just as adding a second head to a floppy disk drive increases the manufacturing cost a minimal amount, so slipping an extra *disk platter* or two into the HDA raises the manufacturing cost by only a small increment. Typical 20 megabyte hard disks will have two disk surfaces, or platters, mounted on the same spindle, and four heads—one for each side of each disk—mounted to a single arm and moving in tandem.

The scheme for numbering the sides of a multi-platter hard disk is simply an extension of that for numbering a double-sided drive: The top of the first platter is side 0, its bottom is side 1, the top of the second disk platter is side 2, and its bottom is side 3. And just as both top-side and bottom-side track 0s on a double-sided drive can collectively be called cylinder 0, so the four track 0s in a two-platter drive can collectively be referred to as cylinder 0.

The amount of data that can be jammed into a single circular track on a hard disk depends on the data encoding scheme used. With current encoding

schemes (mostly variants of a technique called MFM, or modified frequency modulation), 8 to 12 kilobytes per track can be reliably read and written.

Although it would be possible to write and read data from a disk in blocks of 8 or 12 kilobytes at a time, that number was historically considered too large to be practical. Disk controllers, both floppy and hard, are therefore designed to read and write only a segment of a track at a time. The particular number of bytes in each segment, more commonly known as a *sector*, depends on the disk controller hardware and the operating system: The manufacturer designs the disk controller to support several different sector sizes, and the operating system developers choose from among the available sizes. Typically, sector sizes of 128, 256, 512, and 1024 bytes are supported; versions of PC-DOS since 1.0 have used 512-byte sectors exclusively, for both floppy and hard disks.

On a floppy disk, it is possible to squeeze up to ten sectors around a single track and maintain reliability. Conservative engineering, however, led IBM to use only nine sectors per track (and only eight sectors in DOS 1.x); nine sectors per track times forty tracks per side and two sides per disk multiplies out to the now familiar 368,640 bytes per diskette.

With their higher rotational speeds, rigid surfaces, and far more stringent manufacturing tolerances, hard disks can hold both more tracks per side and more sectors per track. The most dramatic improvement comes from cramming in far more tracks per side than is possible on a floppy disk: A typical hard disk may have well over 600 tracks per inch. (As you can imagine, accurately positioning the head over each one of those very thin tracks requires some tight manufacturing tolerances. That's why multiple-platter hard disks are so popular: It costs no more to move six heads at a time across three platters than it does to move two heads across a single platter.)

Hard disks can also support more sectors in a given track, though here the improvement is not nearly as dramatic: The typical hard disk might have 17 sectors per track, while newer disks (using an encoding scheme called Run-Length Limited Coding, or RLL) might have 25 or so.

Higher data capacity is one of the two major advantages hard disks have over floppies; the other is speed.

While dividing the track up into sectors ameliorates certain problems—there's less data to buffer, for one thing—it creates a new problem: Now we not only need to specify a side and track to find a given piece of data, we also need to indicate what sector number within the specified track we want.

Some New Terms

If you read advertisements or spec sheets for hard disks, you're likely to encounter a number of terms used to measure disk performance. In particular, the expressions *seek time, access time, latency,* and *transfer rate* keep popping up in any discussion of disk performance.

Seek time. Seek time is simply the length of time the disk head takes to move from whatever track it happens to be on, to whatever track you want to read. Since that obviously varies each time you read—depending upon where the head was, and how far it has to go to get to the desired track—there are a couple of sub-species of seek time, the most important of which is *track-to-track seek time.*

Track-to-track seek time. Track-to-track seek time is the length of time required to move from one track to an adjacent track. Track-to-track seek times in AT-class disk drives are generally on the order of 8 to 10 milliseconds. Seek times for floppy disks are many times greater than those for hard disks.

Access time. Access time is the time required for the head to *seek* to (i.e., move to) the track that holds the data you want. Clearly, if you left the disk head on

track 3 after your last read and you now want to read data from track 4, the access time would be equal to the track-to-track seek time. But how often will you be lucky enough to have the next track you want to read adjacent to the track you just read? (Actually, more often than you might think. Especially if you're a regular user of Speed Disk.)

Average access time. Average access time is a measure of the time it takes, on average, to move the head from the current position to the track you want to read. At first glance, it might seem that average access time would be equal to the time required to seek across half the number of tracks on the drive. That would be true, *only if* the head were always left at the edge of the disk. But after a disk operation, the head may left anywhere over the disk—the optimal situation would be when it was left in the center of the disk, since then the next access would, on average, be across *half* of a half the total number of tracks, or one-fourth the total.

If you're thinking that the average access time must be somewhere between the time required to seek across one-fourth and one-half the total number of tracks, you're right on target. As it turns out, it can be shown that the average time to access any arbitrary track is equal to the time required to seek across *one-third* of the tracks. (Note, by the way, that this measurement of average access time is independent of the optimizations that can be performed by an efficient OS, which will try to organize data in such a way that data to be read sequentially will be stored on sequential tracks. That means that in real life, much of the time we simply have to move a single track.)

Two requirements must be satisfied before the disk head can begin reading data: First, it must *seek* to the required track; second, the particular sector of the track that is going to be read (or the sector that will be read first, for multiple-sector reads) must spin around under the head. As we've just learned, the

average time required for the head to get to the target track is called the average access time. Once at the right track, the time the head spends dawdling until the right sector comes around is called *latency.*

Average latency. Average latency is the time required for the disk to make half a revolution. Hard disks spin at 3600 rpm, or one revolution every 16.67 milliseconds, so the average latency is approximately 8.3 milliseconds. Since all hard disks spin at the same speed, this figure doesn't vary from drive to drive, so latency time won't help you compare one hard disk with another. But it's interesting to note that floppies, spinning at 300 rpm, have an average latency of more than 100 milliseconds, more than ten times greater than the average latency for hard disks.

Transfer rate. Transfer rate is the speed at which bits are read off the disk—that is, the speed at which they can be transferred from the disk to the computer. Transfer rate depends both on how fast the disk is spinning—you can't read a bit until you get to it—and how densely bits are packed around the track. Here again, hard disks dramatically outperform floppies, both because of the hard disks' higher rotational speed (3600 rpm versus 300) and because of their higher bit-packing density.

Most hard disks today have a transfer rate of 5 megabits per second; with the phasing in of a new hard disk interface standard, known as ESDI, or Enhanced Small Device Interface, transfer rates are in the process of taking a quantum jump to 10 and even 15 megabits per second.

Sector Addressing

Earlier we mentioned an obscure specification of hard disks called the latency time—the amount of time spent waiting for the data we want to rotate around under the head, after we have seeked to the track we want. What we're really waiting for is the particular *sector* we want to come flying by. The question is: How do we recognize that sector when we see it?

Early floppy disks took a brute-force approach to sector identification. Holes were punched at regular intervals around the circumference of the disk, and each hole, which could be mechanically sensed, marked the beginning of a sector. This method was called *hard sectoring.*

Apart from being rather inelegant, hard sectoring didn't adapt well to high-performance drives. A less primitive means of sector identification therefore was developed, one that involved encoding each sector's address into the sector's data. (Sectors are numbered sequentially around the track, so a sector's address is simply its sequence number around the track.) This approach came to be known as *soft sectoring.*

So far, we've been describing strictly physical characteristics of a disk drive: the number of tracks and sides, the size of each sector, access times, and so forth. All of these are characteristics that are determined by the hardware—by the design of the drive itself and that of the controller.

As we've seen, before we can read a sector, we have to put some kind of identifying marker on it. With hard sectoring, holes are punched in the disk, and we use those as a reference in finding each sector. With soft sectoring, before the disk is first used, the sector address of each sector is actually written into the sector itself. (This address stamping takes place during the formatting of the disk.) The address goes into a kind of preamble to the sector data proper. Along with the sector address in the preamble, some special synchronization bytes are written; these are a unique sequence of bytes that appear at the head of each preamble, letting the disk controller know that it's about to read a sector address. And just to make the story complete, we should mention the gap bytes—filler bytes that are placed between sectors to create a timing tolerance for the reading of each sector.

Physical Formatting

Writing the sector addresses, sync bytes, gap bytes, and a few other miscellaneous chunks of data into the sector preamble is called *hard, physical,* or *low-level formatting,* because the job can be done only by hardware in the disk controller. During hard formatting, software in the host machine tells the disk controller to format a track, picks one of the available sector sizes, and specifies a few other parameters; from then on, it's the job of the controller itself to execute the format.

The physical formatting must be done before any soft-sectored disk can be used for data storage. A second, separate process, called *logical formatting* (which we'll talk about in a minute), must also take place before a disk is ready to store data.

What's potentially confusing about the physical and logical formatting processes is the fact that DOS's FORMAT command carries out both aspects of the job for floppy disks, but only the logical formatting for hard disks. When you format a floppy disk, the FORMAT command first performs a physical (hard) format on the disk, followed by a logical format; when you format a hard disk, it skips the physical formatting. It does so, presumably, because most hard disks are sold pre-formatted and never need physical formatting again.

Well, *usually* never need physical formatting again. Let's look at a situation when you might want to physically reformat your hard disk.

Must You Interleave Me?

The major job of a hard format is to write the address of each sector into the preamble of sector data. You may have assumed that sectors are numbered sequentially around the track (I implied as much), and in point of fact, many times they are. But there's no law that says they must be, and there are good reasons *not* to number sectors sequentially.

Suppose you have a computer whose performance is perfectly matched to your hard disk: As data comes

flowing out of the disk controller, the computer always manages to jam the data into memory, getting back to the controller just in time to grab the next batch of data that's been read.

A lovely situation. But in real life, things seldom work out quite so copacetically. Frequently, once a disk has seeked to the track it needs, it can start spitting out data faster than the computer can receive it. What happens is this: The controller reads a sector of data, and the computer stores it away. The computer turns back to the controller, ready for more data. But by now, the next sequential sector is history; the third sequential sector is about to come skimming under the heads. Now it's the computer's turn to wait, while the disk makes nearly a full revolution and the beginning of sector 2 spins around again.

This is about the worst match possible, since it yields only a single sector of data for each revolution of the disk. So let's get tricky. Since the sector addresses are simply data—a special kind of data, but data nonetheless—we can change them: We'll just label every other sector sequentially around the track.

Proceeding around the track, we'll start with sector 1, then we'll skip a sector, then we'll have sector 2, then we'll skip, then sector 3, and so on. When we get back to the beginning, sector 10 will be between sectors 1 and 2, sector 11 between 2 and 4, and so on to sector 9, which would be between sectors 17 and 1 (on a 17-sector disk). We've just changed the *interleave factor* from 1 to 2, and now when we read sequential sectors from the disk, we'll have twice as much time to process a sector before the next one comes flying by.

With an interleave factor of 2, instead of needing seventeen revolutions to get all the data on a seventeen-sector track, we'll require just two revolutions. That's not as good as getting the whole track read in a single revolution, but it's far better than what we had with the original one-to-one interleave.

Changing the interleave factor is one way we can tune the performance of a hard disk to that of its host computer. But beware: As we've just seen (with the initial example of a disk with no interleaving), when you lower the interleave factor to the point where the computer can no longer keep up with the disk, performance suddenly plummets. Changing the interleave factor—and other parameters set by physical formatting, such as sector size—is a delicate operation usually best left to the manufacturer. That's why IBM buries its physical format program on the Advanced Diagnostics disk.

The Software

Getting Logical

So far, we've been dealing with physical characteristics of the disk hardware—the number of tracks or cylinders per platter, the number of platters, the sector size in bytes, the number of sectors, the interleave factor, and so forth. These characteristics are either immutable elements of the hardware, as in the case of the number of platters, or factors set by the hardware according to instructions from the operating system (and thereafter seldom altered). There is a whole other group of disk characteristics, however, that has to do with how the operating system organizes and finds data on the disk.

For speed and efficiency in accessing particular bytes out of a vast sea of 20, 40, or many more megabytes, it is clearly necessary for the operating system to construct some directories and indexes telling it what's where, what parts of the disk are spoken for, what parts are free, and even what parts should never be used because of physical damage. The way such information is organized on the disk is called the logical format of the disk, and the process of writing the various directories and indexes that support this organization is called logical formatting. As mentioned earlier, when you format a hard disk using the DOS FORMAT command (or our Safe Format program), all you are really doing is a logical format; the physical format has already been done. With a floppy disk, on the other hand, the DOS FORMAT command first performs a physical format, then automatically follows with a logical format. Safe Format, on the other hand, will only perform a physical format if it needs to.

No matter what kind of disk we use—hard or floppy, fixed media or replaceable, 360K or 60 megabyte—DOS always uses the same logical format, which

organizes the disk into four main areas: the *boot record*, the *File Allocation Table* (FAT), the *root directory*, and the *data area*. (On a hard disk that can be partitioned among different operating systems there is yet a fifth area called a *Partition Table*; we'll take a look at the Partition Table shortly.)

The Boot Record

The DOS boot record always occupies the first sector of the first track of the first disk side–sector 1, track 0, side 0. (Actually, this is strictly true only of floppy disks, because hard disks reserve the very first sector for the Partition Table.) The boot record does just what its name implies: It lets the computer pull itself by its bootstraps, from empty-minded idiot to efficient manager, by reading in a very short program—the boot code—that in turn reads in the rest of the operating system.

In addition to the small boot program, the boot record contains another item that's vital to the operating system: a list of key characteristics about the disk. Among the items included in this list are the number of bytes per sector, the total number of sectors on the disk, the number of sectors per track, and the number of heads. Because of the importance of this table, the boot record is written to *all* disks during logical formatting, even floppy disks that don't contain the three system files necessary to make a self-booting, or system, disk.

The second and third areas that DOS sets up during logical formatting—the File Allocation Table and the root directory—are used together to keep track of where each file is stored on the disk, which sectors are currently in use storing files, and which sectors are available.

Clusters

Once a disk has been formatted, the smallest chunk of data that the disk controller is physically able to read or write is a single sector, which, for all DOS disks so far, is 512 bytes. DOS could (and in the case of high-density 1.2 megabyte floppies, actually does) keep track of the status of each individual sector—whether it's in use, free, or shouldn't be used because of damage. But the overhead involved in keeping track of disk space on a sector-by-sector basis is pretty high on a large-capacity disk, so DOS deals instead with multi-sector units called *clusters.*

Before we look any closer at clusters and how DOS keeps track of files on a disk, it might be a good idea to take a look at some of the problems DOS is going to encounter when we ask it to create, expand, contract, and delete our files.

Files in Flux

The basic problem with managing computer files is that they're constantly changing (of course, the ability to change computer files easily is also what makes them so attractive). But consider: Suppose we have a file that's 2250 bytes long, and we're writing it to a fresh, unused, 1.2 megabyte floppy disk. The smallest number of bytes that can be read or written from a disk is a single sector, and that is indeed the minimum number of bytes DOS will read or write on a 1.2 megabyte floppy. We'll need 5 sectors to hold our 2250-byte file. The first four sectors will hold 2048 bytes, which means we'll need only 202 bytes of the last sector. Since we can't write a partial sector, DOS will reserve all of the fifth sector for our file, letting the remainder of the last sector go to waste. Our new file will occupy the first five sectors of the disk.

Now let's write a second file that's, say, seven sectors long. It will occupy the next seven sectors of the data area. Now comes the fun: Suppose we want to add 460 bytes to our first file. As you'll recall, only 202 bytes of the last 512-byte sector was in use, so we still have room for 310 bytes there. But that leaves 150

bytes (460 minus the 310 bytes we added at the end of the last sector); where shall we put them?

We can't simply put them in the first sector after our five-sector file, because that sector is already in use by the next file (the seven-sector file). We could move that next file up a sector, but that would take a while. And, besides, consider how inefficient it would be if the next *one megabyte* of sectors were already taken by files: We'd have to move a megabyte of data just to write a lousy 150 bytes. Not a good solution.

The File Allocation Table

A much better solution, the one that DOS uses, is to maintain a table of the status of all sectors. This important table, called the *File Allocation Table*, or FAT, will tell us whether a given sector is free or in use by a file. Now when we need to expand a file, we just look in the FAT to find the next free sector and reserve it for our file. In this example, the first 5 + 7 sectors are taken, so the first free sector is located some 13 sectors from the beginning of the data area. Our file now occupies the first five sectors of the data area, plus the 13th sector of the data area. So our file is *fragmented* now; The beginning portion is stored in contiguous sectors at the beginning of the disk, while the ending portion is elsewhere. This is not a problem, however, provided we can figure out a way to link this ending piece of the file with the beginning piece, so that DOS knows where to find everything.

For the sake of clarity, I just said that DOS keeps a table, called the FAT, that records the status of all sectors on the disk. That's not exactly true. For a 32 megabyte disk, a table that kept track of each sector would be 64K entries long (32 megabytes divided by 512 bytes per sector). Since for fast access the FAT is stored in memory while the disk is in use, large disks would eat up memory pretty fast; besides, it would take a while to scan through a 128K FAT (64K entries, times 2 bytes per entry). For that reason, DOS groups a number of contiguous sectors together, and

deals with them as a unit called a cluster. It's actually clusters, not sectors, that DOS keeps track of in the File Allocation Table.

Since clusters are a fabrication of DOS, DOS can define a cluster to consist of as many sectors as it wants. Rather than settling on a single cluster size, however, DOS varies the size to suit the medium; the cluster size of a given disk is set during logical formatting and does not change thereafter (unless the disk is reformatted). The 30 megabyte AT hard disk is given a cluster size of 4 sectors, or 2048 bytes. The 1.2 megabyte floppy, on the other hand, is formatted with a cluster size of one sector: 512 bytes. That small cluster size reflects a tradeoff of speed for optimal use of available space. It takes longer to deal with disks with smaller clusters, since more entries must be dealt with in the FAT, more read and write operations are required, and so on. On the other hand, the smaller the cluster size, the less space is wasted at the end of each file.

A moment ago, we were looking at a file that was 2250 bytes long and occupied five sectors on a high-capacity disk. Since each cluster on a high-capacity disk consists of a single sector, we only wasted the bytes at the end of the last sector—er, cluster—which amounted to about 200 bytes. But suppose we moved that same file to a 30 megabyte hard disk, formatted with a cluster size of four sectors? DOS uses the FAT to keep track of space on a cluster-by-cluster basis; since a 32 megabyte disk will probably have a four-sector (2048 byte) cluster size (in DOS 3.x), the minimum amount of space DOS can allocate on that disk is 2048 bytes. Our 2250-byte file will take only two clusters when we move it, but since the clusters are now 2048 bytes apiece, the number of wasted bytes at the end of the last cluster has jumped from around 200 to more than 1800.

That unusable space at the end of a cluster is called *slack*, and it's the reason files sometimes seem to shrink when you move them from a hard disk to a

floppy and expand when you move them in the other direction. We accept the greater slack intrinsic to larger cluster sizes in return for the better performance we get with larger clusters.

You can get a report on the percent slack of each of your files, an entire directory, or a whole disk, by using the File Size program in the Norton Utilities. File Size can also determine whether there is room for a group of files to fit on another disk, by taking into account the amount by which files will shrink or expand when moved from disks with one cluster size to disks with another cluster size.

Beefing Up the FAT

Back to the FAT. So far, we've said that DOS maintains a table at the beginning of each disk called a File Allocation Table, or FAT. We know that the FAT is used to keep track of which clusters are free, and which are in use and by what file. We also know that a cluster is simply a convenient grouping of one or more sectors, invented to cut down the amount of bookkeeping necessary to keep track of space usage on a hard disk. What we haven't examined so far is how DOS actually stores that information in the FAT.

The layout of a FAT is simplicity itself. Two bytes (for DOS 3 and later) are allocated for each cluster on the disk. The sequence number of each two-byte entry corresponds directly with the equivalently numbered cluster in the data area. Whoops—we need to back up again: We haven't learned how DOS numbers clusters yet.

Three Ways of Counting

Back when we looked at the physical layout of a disk, we found that the disk controller refers to a particular location using a three-coordinate system: the side number (0 or 1 for a floppy), the track number (0 to 39 for a double-sided, double-density floppy), and the sector number (1 to 9 for a 360K floppy). Since the hardware deals with disks in this three-coordinate system, it's necessary that, at some level, the PC be able to deal with the disk using the same system—which is why all the built-in ROM-BIOS calls specify disk locations via side, track, and sector. For the operating system, however, this three-coordinate system is awkward, what with the number of tracks, sectors, and sides all changing from one disk to the next. DOS therefore locates data on a disk by a one-dimensional sector numbering scheme; DOS simply numbers all the sectors sequentially, starting with sector 1 of side 0, track 0, proceeding through all the sectors on track 0, then on to the sectors on side 1, track 0. All the sectors in a cylinder are numbered first before DOS moves on to the next cylinder; that minimizes head movement, and thus access time, when sectors are read sequentially. The continuous sector-numbering scheme that DOS uses is called *logical sector numbering*; unlike physical sector numbers, which start at 1, logical sector numbers start at 0.

Don't be confused by the statement that DOS numbers the disk sectors sequentially; there's no physical formatting going on here, no addresses actually written on the disk. What's occurring is simply a translation between the way DOS refers to a given sector and the way the BIOS and disk controller refer to it. Think of it as the difference between calling it Sector One in English and Premiere Secteur in French.

Once we've got DOS's sector numbering scheme down pat, the cluster numbering scheme follows pretty quickly. First, we skip over all the sectors occupied by the boot record (always one sector), the

FAT (the size of which varies with each disk format but is constant within each format), and the root directory (which we haven't discussed yet). These three areas, which are always set up during logical formatting, are all considered part of the system area and are excluded from the cluster numbering scheme.

All the sectors beyond the system area, starting with the first sector after the root directory, are considered part of the data area and are included in the cluster numbering scheme. On a disk with a cluster size of four sectors, the first four sectors of the data area will be called cluster 2 (that's right, cluster numbering starts at 2, not 0 or 1), the next four sectors will be cluster 3, the next four will be cluster 4, and so on to the end of the disk. A 30 megabyte disk, with a cluster size of four sectors, would have about 15230 clusters of data area; the last cluster number would be 15231, one larger than the number of clusters, since cluster numbering starts at 2.

Once More, Back to the FAT

With that out of the way, let's return to the FAT once more. We took this last detour when we observed that the FAT contains information about the status of each cluster on the disk, and that there's one FAT entry for each disk cluster. The first two FAT entries are reserved for special information. The third FAT entry holds information about the first cluster (which, to keep us on our toes, is numbered cluster 2). The next entry holds information about the second cluster (cluster 3), and so on to the end, so that the last FAT entry tells us the status of the very last cluster on the disk. (Strictly speaking, some of the details we're describing now pertain to all floppy disks but only to hard disks with a single partition. We'll get to partitions at the end of this section; their presence doesn't change the basic workings of a disk.)

Let's see what information the FAT entry needs to give us. First, since even the best disks can have a few bad sectors, the FAT entry for a given cluster could tell DOS if there's a bad sector anywhere in the

cluster; that way, DOS would know never to allocate that cluster to a file. And indeed, that's one of the pieces of information that DOS records when it creates the FAT during logical formatting. After DOS has finished formatting a disk—writing the boot record and constructing a fresh FAT and root directory—it attempts to read every sector on the disk; clusters that contain sectors that can't be properly read are marked as bad in the FAT. This process is called bad sector mapping, and it's very important to the integrity of our data.

Incidentally, you might be wondering what data DOS could be trying to read off a freshly formatted and supposedly empty floppy disk. A good question, the answer to which is that a "blank" floppy disk is not empty; it's only devoid of data useful to us. When DOS or Safe Format formats a floppy, it writes a special byte (F6) across all of the data area. Later in the formatting process, it tries to read that byte from each sector.

There may be some sectors that the disk controller is unable to read; the controller in this case passes an error message back to DOS, and DOS marks the cluster containing that sector as bad. A few other sectors may be read okay by the controller but have incorrect cyclic redundancy check bytes, indicating there has been an error in reading the data.

Cyclic redundancy check? You may have heard it called the CRC. When a sector is written, a special *checksum* value, called the cyclic redundancy check, is calculated, based on the value of all the bytes written to that sector. This value (also known as a CRC) is written to a special location on the disk immediately following the sector data. When the sector is later read off the disk, its checksum, or CRC, bytes are also read. The same calculation that determined the CRC bytes that were written on the disk following the sector data is performed again, this time on the data that was just read. The CRC calculated from the data just read is compared to the

CRC that was calculated and written to the disk. If the two values don't match, it means the data read is not exactly the same as the data that was written, and therefore that a read error has occurred. The cluster containing that sector is then marked bad.

By the way, you may be surprised to learn that when you tell DOS to Verify a disk copy—using either the /V switch with the COPY command, or the command SET VERIFY ON, DOS does not compare the source data byte-for-byte with the destination data; it simply rereads the destination file and verifies that there are no CRC errors or any other type of read errors. If the CRC checks out, chances are good the data is correct.

But enough of these tangents; let's finish our tour of the FAT. So far, we have seen that we need to keep two facts about each cluster in its corresponding FAT entry—whether the cluster is free or in use by a file and whether the cluster is bad. And if files didn't randomly grow, shrink, or get deleted, that's about all we'd need to keep in the FAT. But remember earlier when we were writing a short five-cluster file at the beginning of a disk, followed by a second, unrelated file? Then when we tried to add some data to our first file, we couldn't grab the next sequential cluster, because it was already in use by the second file; we had to grab a cluster somewhere further down the road. That presents a problem: When DOS tries to read the first file, how will it know where to find that other cluster, the one we wrote further down the disk?

The answer is simplicity itself. We wrote the first five clusters of the file into the first five contiguous clusters on the disk, then skipped some clusters and wrote the file's sixth cluster; why not have the FAT entry for the last contiguous cluster we wrote—the fifth cluster—point around those skipped clusters to the sixth cluster of the file? If the file was written into clusters 2 through 6, and then we added cluster 14, we could write a 14 into the FAT entry for cluster 6. That 14 in

cluster 6's FAT entry would say: "You will find the next portion of this file at cluster location 14 on the disk." For consistency, we would make each FAT entry that's in use by a file point to the next FAT entry in the file, so we would have a *chain* of FAT entries, each entry pointing to the next. And since the sequence number of each FAT entry is the same as the cluster it is reporting on, to locate each cluster in the file, we would just follow its chain in the FAT. This is exactly the procedure that DOS follows.

If you've followed me to this point, you should have a good understanding of the basics of DOS's disk file structure. We just need to fill in a couple of simple details. Such as: Now that we know how to find all the clusters in a file by tracing through its chain in the FAT, how do we locate the beginning of that chain? In our example, the first cluster of our six-cluster file was also the first cluster of the disk's data area—but only because I said it was. How are you going to find that starting cluster tomorrow? And do you remember where I said the second file starts?

Obviously, we need to keep a list somewhere of the starting FAT entry for each file. If we know where the first FAT entry is, we also know where the first file cluster is (they're numbered the same, remember, with each FAT entry reporting on its equivalently numbered cluster), and since each FAT entry points to the next one in the chain, we can find all the clusters in our file.

In addition to the list of starting cluster numbers, which enables us to find the first cluster in each file, we also need a means to find the last cluster in each file. We could put that information in the same place as the starting-cluster information, by listing either the last cluster number or the file's size. Alternatively, we could use the FAT itself, placing a special byte into the FAT entry that corresponds to the last cluster of each file.

In a way, DOS does both: The file directory (which we'll come to presently) records both the length of

each file and its starting cluster number, and the last FAT entry of each file is indeed specially marked.

Our discussion of the all-important FAT is now nearly complete. We know that the FAT is a table of two-byte entries (for DOS 3), with one entry corresponding to each cluster on the disk. (Earlier versions of DOS used 1.5 bytes for each FAT entry. The practical significance of this is discussed at the end of this section.) Each entry tells us whether its associated cluster is available, defective, or already in use by a file. If the cluster is in use, the entry either points to the next cluster/FAT entry of the file, or tells us the cluster is the last cluster of the file. So a FAT entry can contain one of four values:

Value (hex)	Meaning
0000	This cluster is available
0002–FFEF	Cluster in use by file (the number points to the next cluster in the file)
FFF0-FFF6	Reserved; not used
FFF7	Bad cluster; do not use
FFF8–FFFF	This is the last cluster in a file

And we know that there is information in a directory that points to the starting FAT entry for each file.

The Root Directory

The directory—actually the root, or main, directory—is the third and last part of the system area of any disk formatted by DOS. If you've ever looked at a directory using NU (or the new Directory editor that's included in the Advanced Edition of the Norton Utilities), you've probably discovered that each directory entry holds a good deal more information than the starting cluster number and size of each file.

Clearly, in addition to the size and starting cluster number, each directory entry needs to list the filename associated with the starting cluster number.

Looking at it another way, we could say that there is one directory entry for each file on the disk, and that a file's directory contains a pointer to its first entry in the FAT chain, i.e., its first cluster. In addition, each 32-byte directory entry contains fields for the time and date, a one-byte field of *file attributes*, and ten bytes that are reserved by DOS. So the format of each directory entry looks like this:

Description	Size (Bytes)	Format
Filename	8	ASCII characters
Extension	3	ASCII characters
Attributes	1	Each bit represents an attribute
Reserved	10	Unused (so far)
Time	2	Word, coded
Date	2	Word, coded
Starting FAT entry	2	Word
File size	4	Long integer

The attributes represent one of several special properties that can be applied to each file, such as *read-only, hidden, system, volume label, subdirectory*, and *archive*. (The Norton utility File Attribute can set and reset file attributes.) The attributes are bit-encoded; a given attribute is set if its associated bit is a 1.

The volume label attribute presents an unusual situation. No file actually exists for a directory entry with this attribute. Only one directory entry in a disk should have the volume label attribute, and that entry should be in the disk's root directory. The label is stored in the filename and extension fields of the directory entry, which in this case are treated as a single field.

Before we get too wrapped up in the innards of directories, let's take a moment to distinguish

between the root directory and subdirectories. So far, we've been talking about the root directory as if it were the only directory on the disk, and indeed, the root directory is special in a couple of ways. First, the root directory is the only directory that is not in the disk's data area: The root directory is the third element of the system area, and it's always located immediately following the FAT. The size and location of the root directory are fixed; they are established during logical formatting and cannot be changed afterward. Its fixed size also differentiates the root directory from subdirectories, which can be created, grow, shrink, and be deleted as necessary.

The size of the root directory varies with the type of disk. On a 360K floppy, FORMAT creates a root directory with room for 112 entries; on the AT's 30 megabyte hard disk, FORMAT creates a root directory that can hold 512 entries. That means, for example, that if you want more than 512 files on a 32 megabyte hard disk, you'll have to create some subdirectories.

Any entry in the root directory can refer to either a file or a subdirectory. Subdirectories are hybrids. They are assigned space exactly as though they were files. They can grow, expand, and be deleted like a file. But, rather than holding our data, they hold other filenames. You might say that in form, a subdirectory is just like any other file, and in function it's exactly like the root directory. Because subdirectories are stored in the data area, they can be assigned space on an as-needed basis.

A while ago, I said each directory entry contained, among other items, the filename, file size, date and time of file creation (or last modification), and the starting cluster number (FAT entry) of the file. Let's take a closer look at the filename field of a directory entry.

As you might expect, the filename field is an eleven-byte area, divided into an eight-byte name field and a three-byte extension field. Filenames of fewer than

eight characters are padded to the right with blanks; immediately following the eight-byte filename is the extension, padded with spaces to three bytes. (The dot isn't stored; it's assumed to be between the eighth and ninth characters.) If you use the Directory Editor or NU to look at a filename, you will see it is always stored in CAPITAL letters; if you ever change the filename in a directory by means of NU or the Directory Editor, be sure to use CAPITAL letters, because DOS will choke on lowercase letters in a filename.

Just as a FAT entry can either be a pointer to the next entry in the chain, or one of three special codes, so the first byte of the filename field can either be the first letter of the filename, or one of three special codes.

A 0 as the first byte of the filename indicates a completely unused directory entry. Using a 0 to indicate virgin directory entries enables DOS to know when it has reached the end of active directory entries, without searching to the end of the directory. (This convention was not implemented in DOS 1.x.)

A period character as the first byte of the filename indicates that the entry is reserved by DOS, for use in navigating around the directory structure.

Last, but definitely not least, the first byte of the filename may contain a lowercase Greek sigma character. This special marker (whose ASCII value is 229 decimal, E5 hex) indicates that the file has been erased. Most programs, UnErase and Quick UnErase included, represent the sigma character as a question mark, to indicate that it represents an erased and now unknown character.

Fortunately for us and for the Norton Utilities UnErase program, DOS is pretty lazy when it comes to erasing a file. Rather than erasing the actual file data, it simply marks the first byte of the filename with an E5, to indicate that the file has been erased, then clears (zeros) the file's entries in the FAT. (Recall

that a value of 0 in the FAT entry indicates that the cluster is available.) Since it doesn't erase the actual data, or even the starting cluster number in the directory, we can find and recover the first cluster very easily, provided it hasn't been overwritten by another file.

Using knowledge about the structure of the disk, directory, and FAT, the Norton Utilities Quick UnErase program can often recover the rest of the clusters of an erased file automatically; the utility even knows how many clusters to look for, since the erased file's length in the directory is not overwritten. Sometimes the file is so badly fragmented that Quick UnErase either can't locate the right clusters or finds the right clusters but puts them together in the wrong order; that's the sort of problem that UnErase, with its more sophisticated capabilities, is designed to handle. Working with what you know about your file and what UnErase knows about your disk, you can frequently recover even badly fragmented and partially overwritten files.

Tying Up the Loose Ends

So far, we've been talking about two main areas the disk is divided into—the system area and the data area—and the three subdivisions of the system area. To wit, we said the system area is divided into a boot record, which is always the first sector of the disk; the FAT, which directly follows the boot record; and the root directory, which immediately follows the FAT.

This description of the system area is 100 percent accurate for floppy disks and other special disks that can't be *partitioned* for use by multiple operating systems.

High-capacity hard disks are valuable and expensive resources, so, good neighbor that it is, DOS provides a way to share such disks among different operating systems. The trick is to set up multiple partitions, with one partition for each operating system. (It is possible to set up multiple partitions all running the same operating system, but there's little reason to do so,

unless you're running a version of DOS up to version 3.2 and you have a disk larger than 32 megabytes, which is the largest capacity disk such DOS versions can handle; Compaq DOS 3.31 and PC-DOS 4.0, on the other hand, can support partitions greater than 32 megabytes.)

Since DOS couldn't get the time of day from a Unix partition, and Unix doesn't know how to read a DOS partition, it follows that there must be some area of the disk common to all partitions. This common area, known as the Partition Table, specifies the disk location and length of each operating system's partition.

The Partition Table is always found in the first sector of any partitionable disk (just as the boot record is found in the first sector of any nonpartitionable disk). On a partitionable disk, the Partition Table must be set up before logical formatting. (You may recall that logical formatting is the only type of formatting the DOS FORMAT command or Safe Format does on a hard disk; physical formatting of a hard disk is generally done by the manufacturer.) The DOS command FDISK sets up the Partition Table. That's *all* it does, as a matter of fact. Before you can use a hard disk, then, you must first run FDISK, then run FORMAT or Safe Format.

If the partitioned disk is also the boot disk, the question arises: With multiple operating systems residing on the same disk, how does the PC know which system to start up at boot time? The answer is: A record of the currently active partition is also kept in the Partition Table.

Several other miscellaneous housekeeping items are stored in the Partition Table, but only one of them is of any real interest to us. We said that on a non-partitionable disk, the first sector is always the boot record, a short program that loads in the rest of DOS. Since DOS expects to find a boot program in the first sector, partitionable disks oblige by also putting a

boot record in the first sector, as part of the Partition Table.

In a case like this, when the boot record (boot program) is part of a Partition Table, it's actually called a *master boot block*. Now when we boot off our partitionable disk, we'll first read in the code in the master boot block of the Partition Table; that code will in turn find out which partition is active and read the boot code from that partition. So if the DOS partition occupied the entire disk, the master boot code would read in the DOS boot record, which would in turn read in the rest of DOS. If there were two partitions, and Unix were active, the master boot code would read in the boot code from the Unix partition, which, being partial to Unix, would read in the rest of Unix.

That's all there is to the Partition Table. The remainder of a partitionable disk is organized exactly as described above, except that now our description of the organization of a DOS disk applies only to the DOS partition. In the typical case, where the entire disk is given over to the DOS partition, the only difference would be that everything is "slid forward" one track, to make way for the Partition Table. (The Partition Table uses only the first sector of track 0, but the remaining sectors on the track are skipped anyway.) The DOS boot record would still start at sector 1, but of track 1; the FAT would start at sector 2, track 1, and so on.

Speaking of the FAT, let's confuse one more item that we simplified during our discussion. You'll recall that we've been talking about two-byte FAT entries; two bytes being 16 bits, and 16 bits being capable of holding a number as large as 65,535, a two-byte FAT entry should be good for disks with up to 65,536 clusters.

Alas for our discussion, the FAT with two-byte entries is a Johnny-come-lately. Until DOS 3.0, all FAT entries were 1.5 bytes. 1.5 bytes, or 12 bits, can express numbers up to only 4095, which is a relatively

small number of clusters. Organize a large hard disk with a small number of clusters, and you've got to make each cluster a large number of sectors—which, unless your files are gargantuan, is inefficient.

As of version 3.0, therefore, DOS allows for FATs with two-byte entries on large disks, while continuing to use 1.5 byte FATs on smaller media. Whether its entries are 1.5 or 2 bytes long, the FAT still works the same, so the practical significance of having two different-sized FATs is nil, unless you like doing FAT arithmetic (which you won't need to do if you're using the FAT editor in the Norton Utilities Advanced Edition; it does the FAT arithmetic for you).

If you are using the FAT editor, I should mention one more thing: The FAT is so important, DOS stores two identical copies of it.

Finally, let me take care of one more little item. You've probably heard about the 32-megabyte file size limit imposed by DOS, and you may have wondered where it came from. Let's do a little arithmetic. DOS, you recall, transforms the physical side/track/sector three-dimensional numbering scheme used by the disk controller and the BIOS into a single, sequential, one-dimensional sector number. DOS uses a two-byte word to specify that sector number, so the largest sector number it can handle is 65,535, the largest number that two bytes can express. If we've got 65,536 sectors (DOS logical sector numbering starts at 0, even though physical sector numbering starts at 1), and each sector is 512 bytes, then the maximum number of bytes we can address is 65,536 times 512, or 32 megabytes.

A way around the 32-megabyte limitation is to increase the sector size. With a sector size of 1024, for example, we could address up to 64 megabytes. This is the tack taken by the distributors of some of the immense hard disks now being offered for the PC. Compaq's DOS 3.31 was the first to surpass this limitation by increasing the two-byte number

specifying a DOS sector to four bytes. Previously, the two-byte specification allowed up to 65,536 sectors. Using four bytes, on the other hand, allows up to 4 *billion* sectors (What famous astronomer does that remind you of?); therefore, this version of DOS can handle partitions greater than 32 megabytes. PC-DOS and MS-DOS 4.0 also support partitions larger than 32 megabytes using this same scheme.

Things That Go Bump in the Disk

Now that we know how a disk should work, we are in a pretty good position to understand the kind of things that can go wrong—and more important—how to avoid or fix them.

Between the directory and FAT, it's pretty easy to imagine some of the things that can go awry. It's possible, for example, for a cluster in the FAT to be marked as in use, yet not be part of any file's allocation chain. Such an anomaly would signal an orphaned cluster (sometimes called a *lost cluster*). Getting more imaginative, the FAT entry for one cluster could point back to the entry that just pointed forward to this entry, creating a circular chain. Or the allocation chains for two or more entries could both point to the same cluster, meaning a single cluster had been allocated to two different files—a mistake referred to as a *cross-linked* file.

Such errors can happen either when the moon is full and strange creatures roam the night, or when you turn off your machine while a program still has a file opened or is writing to the disk. Quit your programs before shutting down.

In any case, the The Norton Disk Doctor utility can spot, report, and fix logical errors such as these, all with a minimum of intervention from the user. In this way, it's far superior to the DOS CHKDSK program. In fact, CHKDSK will cheerfully mislead you if you run it without the /F switch, by saying that it fixed the errors; don't believe it.

What's more, CHKDSK doesn't check for, and can't fix, physical disk errors, such as sectors that can't be

read. Disk Doctor can handle these sorts of problems. The program can reformat bad sectors and write back the old data. In addition, Disk Doctor can detect and fix bad Partition Tables, detect and fix bad boot records (making a disk bootable again), detect and correct problems with the FAT, and check out and correct the root directory.

In a way, Disk Doctor represents a translation of the knowledge in this booklet into a very powerful program. Disk Doctor is designed to give even a relative novice the power to correct a variety of problems with a minimum of effort. Even though you may not need to know much about how your disks work in order to keep them in top shape, however, a little inside information never hurt anybody. Except maybe a few unlucky souls on Wall Street...

Conclusion

If you've come this far, you should have a pretty good understanding of what's going on behind that little red disk light. We haven't discussed everything there is to know about disks, of course, but we have covered all the important concepts and terms you're likely to run into in your day-to-day dealings with your PC and DOS.

If you should decide to test out some of the material you've learned, poke about your disk, maybe even discover a few items I didn't have space to mention here, I hope you'll discover that the Norton Utilities are as much fun for exploring your disk as they are useful for recovering data.

Index

A-C

D-F

G-N

P-R

S-Z

NOTES

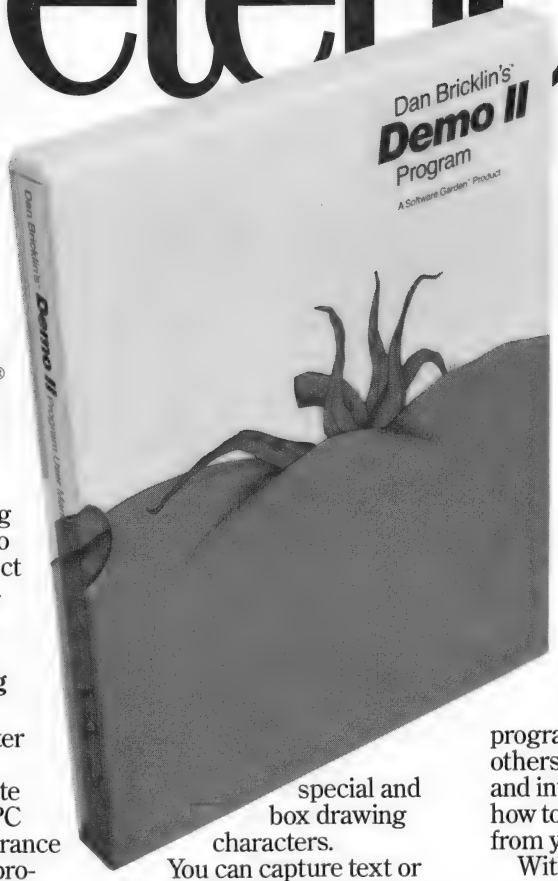
NOTES

NOTES

NOTES

NOTES

The Great Pretender.



Did you know that many of the world's most successful software publishers have been faking it for years?

That's right. Lotus,® Ashton-Tate,® WordPerfect,® Symantec,™ —yes, even Peter Norton Computing.

We've all been using Dan Bricklin's™ Demo II™ Program to concoct the world's most convincing prototypes, demos and tutorials.

The next best thing to being ware.

Now published by Peter Norton Computing, Demo II lets you create slide shows on your PC that mimic the appearance of an actual running program, complete with menus, screen images, even grinding disk drives.

It's so realistic that, if you didn't know better, you'd bet your ASCII you were seeing the genuine article.

You can create text screens with all 256 characters and attributes, including the special and box drawing characters.

You can capture text or bit-mapped graphics images from other programs.

You can use the overlay facility to merge elements from existing slides into new ones, or to create special effects.

You can duplicate *any* user interface, from the pull-down menu to the Lotus light-bar.

(Not to mention the one you just thought of.)

You can take advantage of over 100 run actions to control the sequence of slides, respond to keystrokes, produce sounds and a whole lot more.

And you can run off copies of the run-time until you run out of floppies.

A picture is worth a thousand lines.

With Demo II at your disposal, you can conceptualize your programs, describe them to others, refine the functionality and interface and teach users how to get the best results from your end result.

Without writing a single line of code.

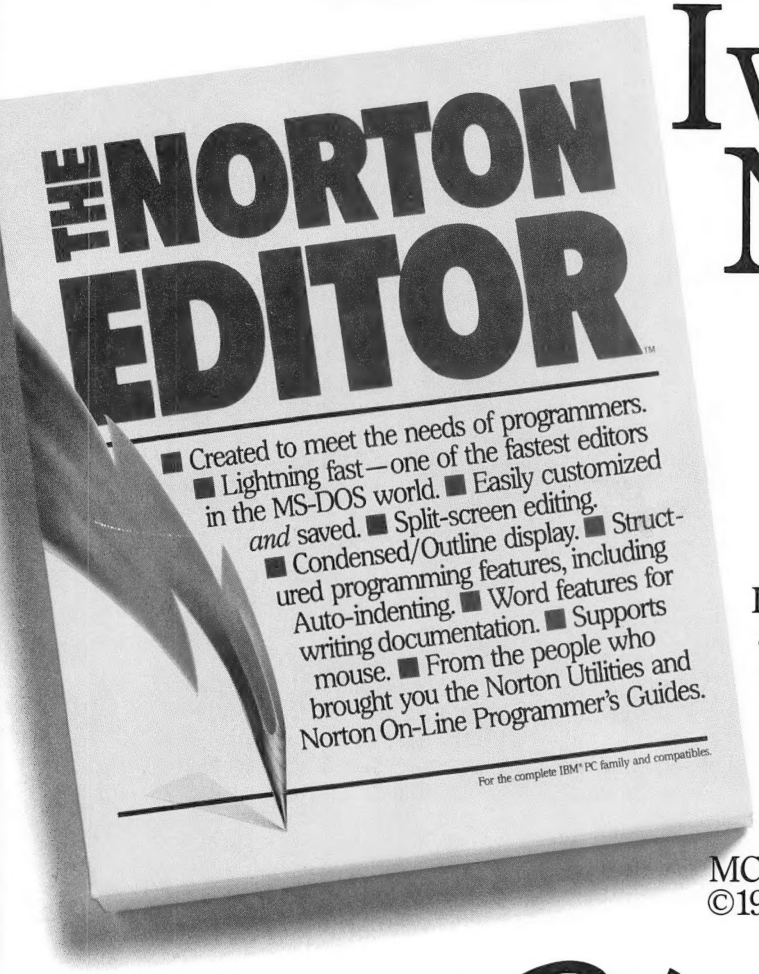
Which means that, when you *do* get around to writing it, you'll only have to write it once.

Or you'll only have yourself to blame.

Peter Norton COMPUTING

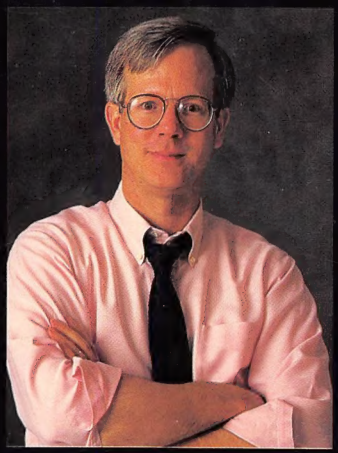
Designed for the IBM® PS/2® and PC families and 100% compatibles. Available at most software dealers or direct from Peter Norton Computing, Inc., 100 Wilshire Blvd., 9th Floor, Santa Monica, CA 90401-1104. 1-800-365-1010. Visa and MasterCard welcome.

"This is the programmer's editor I wish I'd had when I wrote the Norton Utilities."



Designed for the IBM®PC, PC-AT and DOS compatibles. Available at most software dealers, or direct from Peter Norton Computing, Inc., 2210 Wilshire Blvd. #186, Santa Monica, CA 90403. To order: 800-365-1010 (Visa and MasterCard welcome). 213-319-2000. MCI Mail: PNCI. Fax 213-458-2048.

Peter Norton
COMPUTING



Peter Norton

A product of Peter Norton Computing, Inc.
100 Wilshire Boulevard, 9th Floor, Santa Monica, CA 90401
Customer Service: 213-319-2010
Technical Support: 213-319-2020

AE 20510